JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2003**

Vol. 2, No. 4, July-August 2003

Use Cases and Aspects – Working Seamlessly Together

Ivar Jacobson

Abstract

Aspect oriented programming (AOP) is "the missing link" to allow you slice a system, use case by use case, over "all" lifecycle models. This will dramatically change the way complex systems are understood, how new features are added to systems, and how systems are implemented and tested. AOP will also add a new dimension of reuse to software development. And it is here to be harvested—now.

1 INTRODUCTION

Use cases have been universally adopted for requirements specification. Use cases start in requirements, are translated into collaborations in analysis and design, and to test cases in test—this is the central idea behind use case driven development. We can conceptualize a system as a loaf of bread and cut it into slices. With use cases we can cut the system into use-case slices with elements from each lifecycle model—almost! It is almost true since today (a) the coding of a component or a class requires us to merge the code derived from several use cases so the individual slices will be dissolved and are no longer recognizable, and (b) the extension mechanism supported when working with use cases (<<extend>>, etc.) with the current UML is neither supported between analysis and design elements such as collaborations, components, classes, nor in "traditional" implementation environments such as Java or C#. The root problem is the limitations of the currently used languages.

Aspect-oriented programming or AOP (herein refers to a general implementation technique) is "the missing link." It will allow us to slice a system cleanly, use case by use case, over many models (use-case model, analysis model, design model, implementation model, etc.) so that use cases remain separate all the way down to code. Thus we achieve separation of "concerns." In fact, *we get the separation by introducing slices, we'll call them use-case modules, that crosscut the component modules. We'll later recompose or weave these slices back into a consistent whole—the deployed system*. AOP is based on ideas that are very similar to our extensibility mechanisms, part of which are now supported in the UML with extensions on use cases. Thus, thanks to AOP these ideas are here to be harvested—now.

Therefore, alternative titles for this paper could have been:

Extensions in UML ≈ Aspects in AOP

if the paper was focused on making extensions to an oblivious base, or

AOSD with Use Cases

(AOSD is Aspect-oriented software development), if we wanted to describe a broader technology covering the whole lifecycle with separation of concerns in general.

But our motivation is to add aspects to use cases and thereby be able to slice the system use case by use case over all concerned software lifecycle models.

At Ericsson, we successfully designed a system that could be modified for years to come; a system comprised entirely of components interconnected through well-defined interfaces. We met our customers' demands by configuring the system using components. New features or use cases were usually provided by adding a new component and changing some existing components. We had a product, a telecommunications switching system, considered superior to any other competitive product. Everyone seemed to be happy. That was in 1978.

However, I was not satisfied for the following reasons:

1. A component usually contained not only code to realize a piece of a dominant use case,[1] but also (usually small) pieces of many other use cases.
2. A use case was usually realized by code allocated to several interconnected components.

These two decomposition effects are now referred to as tangling *and* scattering *respectively* (see Peri Tarr, et. al.[1]). These effects were well known, and in 1967 were used as ammunition against component technology. Fortunately, components won for well-known reasons. Still we had to accept and deal with tangling and scattering. Every time a use case was modified or a new use case was introduced, we had to change several components. Similarly, every time the underlying system software was upgraded (e.g., by upgrading the recovery mechanism to make recovery more fine-grained, or by adding a logging capability), we had to make small changes in many components.

This paper describes an approach to attack this major software problem. The "solution" has been underway for many years, but we have not yet been able to nail it down all the way to implementation. Thanks to a new programming paradigm, AOP, "the

[1] In these early days I used the term "function" with a meaning similar to "use case," but, for simplicity, in this paper I have translated "function" to "use case." I first introduced the term use case in 1986.

missing link" has become available and we can "close the loop." *Our long-lasting work on component-based development and object-oriented programming can take another big leap forward*.

I use the term AOP[2], to mean a general-purpose technique as opposed to a specific programming language,[2] [3] [4] and it is "a gift from above" as you will soon see. This general AOP is part of a general software engineering technique AOSD.[5] And, (sorry to burden you with all these terms) to complete the picture, *use case driven developmen* [6] is an approach for AOSD, but an approach that has a long tradition—one that is now being refined. But more about this later, first, let's understand the problem better.

2 ARTICULATING THE PROBLEM

To get some real metrics behind my 1978 critique, I conducted a small case study of a telecom switching system consisting of hundreds of subsystems. Most of them were reusable for different customers (usually entire countries at that time), but typically each customer had their own sets of communication protocols. Therefore we designed customer-specific subsystems for each protocol, in fact two subsystems: one for incoming calls to the switch, and one for outgoing calls from the switch. In my study I selected a subsystem for outgoing calls and found the following:

- The subsystem's base function was to realize a part of the use case Make Telephone Call Using Protocol X. That use case was realized by many subsystems, but the protocol-specific part was allocated to the subsystem I studied. Interestingly just 40% of the code in the subsystem was there to realize the base function.
- The rest of the code realized *small parts of 23 other use cases*, for example, code to block a telephone line using the protocol, code to supervise the alarm level on a group of telephone lines using the protocol, code to measure the traffic over these lines, code to restart the lines in case a software error occurred, and code to support the distribution of the subsystem over several computational nodes.
- About 80% of this code was coded using templates. The component designer (programmer) only knew that these templates had to be used, but did not have an in-depth understanding of their purpose—not very creative work, to say the least.
- The expectation was that the number of new use cases that would be part of the subsystem would continue to grow even more in the future.
- The report was received with interest but the reaction was that this is the nature of software—use cases cross components—and there is nothing to be done about it. I disagreed.

[2] Unfortunately, there is no single reference to AOP or AOSD in the same way as there is no single reference to OOP (object-oriented programming) or CBD (component-based development). These terms are umbrellas for a variety of ideas. For instance AspectJ and HyperJ are AOP languages, and MDSOC and use case driven development are AOSD approaches.

Going through the case study, which had the small parts of 23 other use cases, I made some notes:

- **12 use case parts were noninvasive:** these parts were very simple additions to the subsystem: they could be added without changing the behavior of any other part; they just needed read access to the objects shared with other parts.
- **8 use case parts were extensions to but did not change the base use case:** these parts required access to the execution sequence in the base use case part (the telephone call use case). When the base use case was executed and passed a specific point in its code, these use case parts needed to be invoked. The execution would always return to the point of invocation.
- **3 use case parts had major impact on the base use case:** these parts needed write access to the shared objects, and they would also change the execution sequence of the base use case part.

The solution seemed obvious:

> ***If we could keep use cases separate, even while they cross several components, and maintain that separation all the way down through all lifecycle activities*** *from requirements to test via analysis, design, implementation and testing, and, yes, also in runtime,* ***we would get a system that was dramatically simpler to understand, to change, and to maintain****.*

But how will we get there? I'll try to explain in this paper, which is structured in three parts:

- **Part I: The Basic Idea** introduces the principal solution.
- **Part II: Today—Working with Use Cases** briefly presents the state of the art of use case driven development and the problems we face with incomplete capability for separation of use cases—no aspect support.
- **Part III: Tomorrow—Working with Use Cases and Aspects** describes how we will work with use cases tomorrow when use cases can be kept separate throughout the lifecycle—with aspect support.

3 PART I: THE BASIC IDEA

Keeping Use Cases Separate All the Way Down to Code Meant Use-Case Modularity

At Ericsson, our components provided what I would call *component modularity*. Components work well to provide the whole picture of a system, to describe its *static structure*. They help us understand, design, implement, distribute, test, and configure the system in a way that is analogous to the way we humans have traditionally organized complex problems—by hierarchical decomposition. However*, components didn't help to describe what the static structure was* doing—*its functional behavior—for this we had use cases*. But the problem was that the use cases were not really separated: they cut across components. We tried to specify and design them as separate units, however, when implementing the use cases, they were integrated to a mass from which it was impossible to identify which use case was being implemented by which piece of code. Or, in other words, **the use cases were dissolved into the code, and distilling them from the code was far from easy.**

I was looking for a new kind of modularity to live alongside the component modularity. The new kinds of modules would cross components, and they would be composed with other modules of the same kind to provide the complete functional behavior of the system. Composition would occur on all levels, both inside a component and over all components as a whole. I asked for functional modularity, but that was before the days of use cases—today I would have called it *use-case modularity* [7] . Use-case modules are important so I will discuss them later in the section *What We Get with Aspects*.

To achieve use-case modularity, I needed two mechanisms:

1. **A *use-case separation* mechanism**

   Use cases are designed to slice a system into separate usage related parts. This works very well for use cases that are peers: none of the peers is more basic than the others; none is more mandatory or more optional than the others. An example is a telephone call, which is a basic use case in a telecom system. There are many kinds of calls: local calls, calls to another area or another country, calls from another area or country; all these different kinds of calls are *peer use cases*.

   However, some use cases depend on other, more basic, use cases to work. To separate these use cases we needed an *extension* mechanism. This mechanism would allow a complex system to be developed (analysis, design, implementation, integration, and test) by starting with a *base use case* and then successively extending the base with more behavior without having to change the base—these are called *nonintrusive* extensions. First describe the basic behavior, then add the extra behavior, that is, behavior that is not needed to understand the basic behavior.[3] The goal was to get easy-to-understand design and code by structuring them from a base and let the system grow without cluttering the base with statements that had nothing to do with the base, even if the statements were important for the additional behavior.

   This extension mechanism would, when applied to use cases, give us *extension use cases*. On top of a base including use cases, we would add extension use cases, and when composing the two we got a new base with new extension use cases, and so on. With this extension mechanism we were able to keep most use cases separate all the way down to code and even to executables.

[3] This kind of extension mechanism was not just a technique for describing optional behavior (optional, that is, from the customer's point of view), they were also intended for describing mandatory behavior in a structured way.

2. **A *use-case composition* mechanism**

   For the system to work, we need to compose or integrate the slices, that is, the separated use cases, into a consistent whole to get executable code. We had several options: the weaving together could occur, for example, at precompile time, at compile time, or at execution time.

   Composing *extension use cases* with their base, given the extension mechanisms that I worked with at that time, was relatively straightforward. However, we also needed to compose *peer use cases* that were not separated through extensions. Here we needed to compose use cases with overlapping behavior, for instance two peer use cases may have two operations that are similar but not identical. This is a more complex problem. It is also a more general problem since composing extension use cases is just a special case of composing peer use cases.

Of the two mechanisms, *I prioritized the ability to separate use cases through extension mechanisms, since that would allow us to keep use cases separate down to the level of a component's code*. Integrating the separate use cases could be done by a component developer. It was also the mechanism that very simply could be introduced by adding a precompiler to the development environment. Furthermore, in my experience even with this very small technological change has a big payback: many new features required only simple design extensions, and these kinds of extensions could be tested in a dramatically simpler way. We would get "the most bucks for the money."

Thus my work became focused on extensions for both the use-case separation mechanism and the use-case composition mechanism, and I would leave the much more complex work on composing use-case peers for the future.

First, the Original Extension idea

To explain the idea, I used the following example. The following italics text is a direct quotation from the 1979 paper[7] with a few nonsignificant modifications[4].

> *Our ... language constructs must be supplemented with a possibility to explicitly change the "flow-of-control". We will illustrate this with an example; first how we are doing this today [that is in 1979] and then a possible further development:*
>
> Example: *Assume that we have two use cases, the Call Handling and the Traffic Recording use case. You could consider the existence of the former to be independent of the latter, but not the opposite. However, to construct the Call Handling use case you must also have access to knowledge about the Traffic Recording use case.*

[4] In the 1979 paper the term *function* has been replaced by the term *use case*. The term *reference point* has been replaced by *extension point*. I have used [ ] for simple explanations, and I have used "…" to replace irrelevant text.

The result could look like this (very simplified)[5]*:*

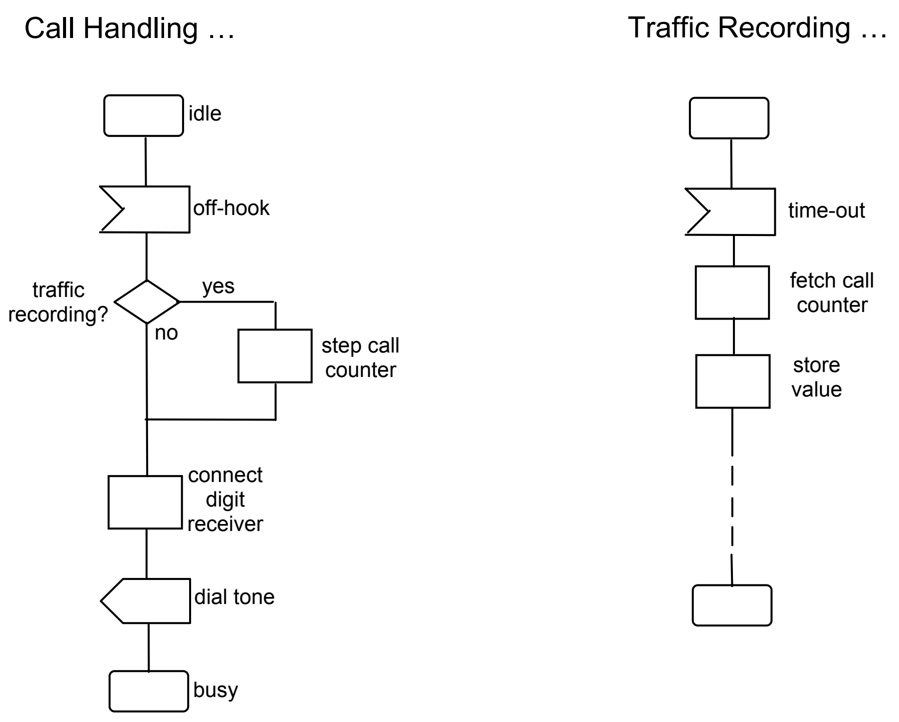

Figure 1: The Call Handling use case cannot be oblivious of activities (e.g. 'step call counter') requested by the Traffic Recording use case[6]*.*

[5] For a reader unfamiliar with telecom, a Call Handling use-case instance is invoked when a calling subscriber takes the phone off the hook: an off-hook signal is sent from the phone to the use-case instance. The use-case instance checks whether traffic recording is requested, if this is the case a counter is stepped. This counter is stepped if a call is ongoing, and it will subsequently be stepped down when the call is terminated. The next actions are "connect digit receiver," "send dial tone" to the calling subscriber, and for this example we don't need to go further.
The other use case is Traffic Recording, the goal of which is to measure the average traffic from subscribers during, say, a 15 minute period. To do that it will have two flows, only one of which will be shown in the diagram in Figure 1. First, every 10 ms it will visit the set of call counters in the system. It will count the numbers that are stepped and divide that number with the total number of counters—the resulting number is a snapshot of the traffic during a 10 ms period. This number is recorded and so are all similar numbers during a 15 minute period. The other flow will calculate the average traffic over the period in question.
[6] The notation in Figure 1 was used when working with the AXE product. It made it into SDL in 1976 and with insignificant changes now also in the UML. Instead of representing an activity as a box, the UML uses a box with round ends. It may be less known, but the input and output boxes are legal UML notation.

In the call handling use case, we were forced to include use case parts that are related to traffic recording. This is unfortunate; the use cases should be kept apart.

By using a simple technique… this can be avoided.

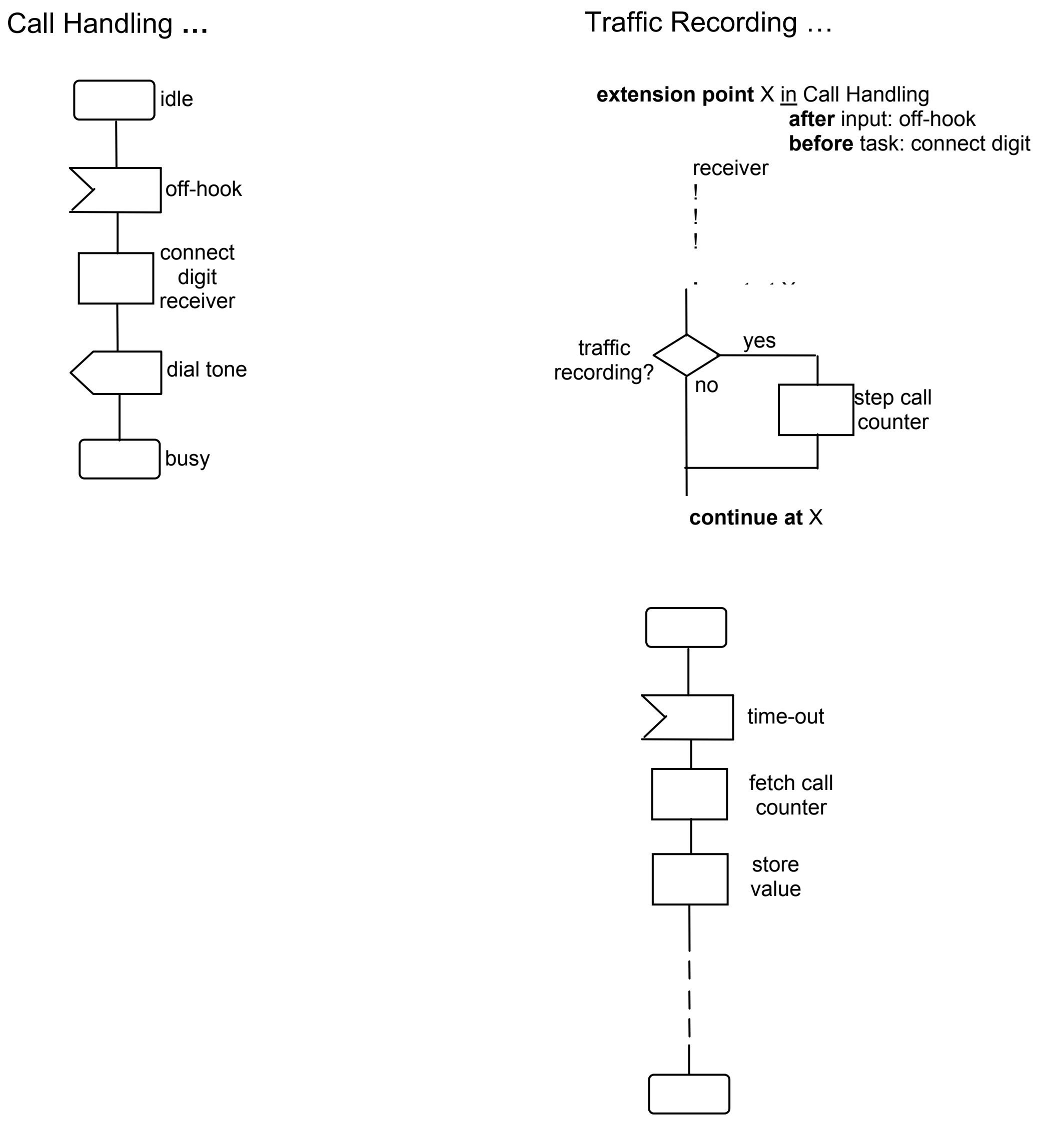

Figure 2: With a simple technique—separation of concerns with extensions and extension points—the Call Handling use case could become oblivious of the Traffic Recording use case.

Thus, we have introduced a possibility to unambiguously refer to another use case description and to change the flow-of-control of a use case. Call Handling and Traffic Recording can be described with use case modularity.

From the same paper [7]:

"Before the execution of a statement in a use case description—which has been compiled into target code—we assume that the micro program (simultaneously) checks whether another use case in an outer layer has a reference to the current instruction address. If so, the execution of the current use case description is interrupted and the sequence that is inserted by the referring use case description is executed."

The microprogrammable implementation of the idea resulted in a patent application [8], which, however, was not approved for reasons I will tell in a moment.

Since a large class of extensions could be safely introduced without intruding on the base, regression testing would, with proper tooling, not be needed for this class. This was expected to result in huge savings in test effort and expense.

To achieve this mechanism I introduced a few quite simple constructs:

- An **extension point** to allow us "to unambiguously refer to another use-case description" by using **before** or **after** declarations into that description. The term "description" meant a diagram or piece of code.
- Two statements, **insert at** <extension point> and **continue at** <extension point>, allow us to extend a diagram or piece of code without explicitly stating this in the base.

In "Use Case Modularity" [7] I wrote: "The technology used to accomplish this is surprisingly simple in principle: Editing in runtime." I recognized that the proposal was just the beginning of a new technique that, once adopted, would evolve on its own.

A few years later (in 1986) I generalized the notion of extensions [9] (see italics below). I apologize for the rather lengthy quotation from the 1986 paper, but it is still valid[7]. I introduced the invented word "**existion**" as an existing set of objects (a base) and contrasted it to extensions.

There are only two kinds of relations between an existion and an extension.

- o *The extension requires no accesses or read accesses only to an existion.*
- o *The extension requires control access (i.e., the extension causes additional instructions to be executed) from an existion, without changing the existion.*

[7] I have replaced the term *probe* with the term *extension point* which is of no significance for the meaning of the idea. I also removed some insignificant, context-specific text marked "..."

The first case means that the extension can use existing actions on an object instance provided these actions do not change the state of the object instance.

The second case means that while an existion is executed atomically, the extension may 'intervene' at specified points. When the extension is executed, the control will be returned to the existion which now continues its atomic action[8]. *More than one extension may intervene in the execution of an existion. An extension point specifies where the intervention is required in the execution of an existion.*

We must provide such constructs so that the extension can be described without changing the existion[9] *...*

The idea is to provide an extension ... with a list of extension points (Figure 3). An extension point specifies an insertion point... During interpretation of a transition path, ... an object instance in the existion allows the desired statements of the extension ... to intervene.

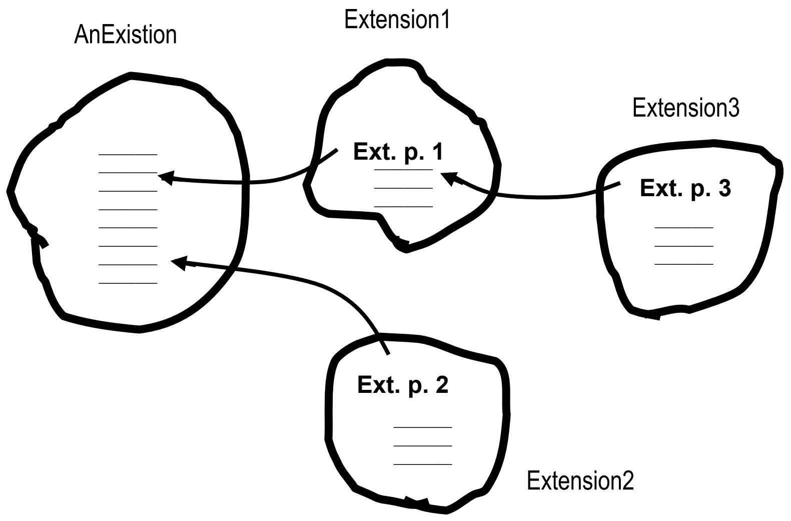

Figure 3: Functional extensions.

An extension may itself be treated as an existion and be intervened by another extension.

Since functional extensions do not change the behavior of existing services, these changes can be introduced in a single step.

Linguistically, an extension can at first glance be viewed as a new class inheriting its existion. ...

[8] The execution of the extension occurs within the atomic action triggered by the existion.
[9] The existion is *oblivious* to the extension.

However, class inheritance is not the phenomenon desired here. Instead, an extension must exist together with its existion; it always requires its existion to be installed and it will only be executed when its existion is executed.

This was 1986. What happened after that? I was of course very excited about the whole idea of adding use-case modularity on top of the existing component modularity. I saw the component structure as a base onto which we could add use-case modules—one after the other (intuitively as shown in Figure 4). More precisely each minor picture in this figure shows how use cases from a use-case model are realized by parts in several components interacting with one another.

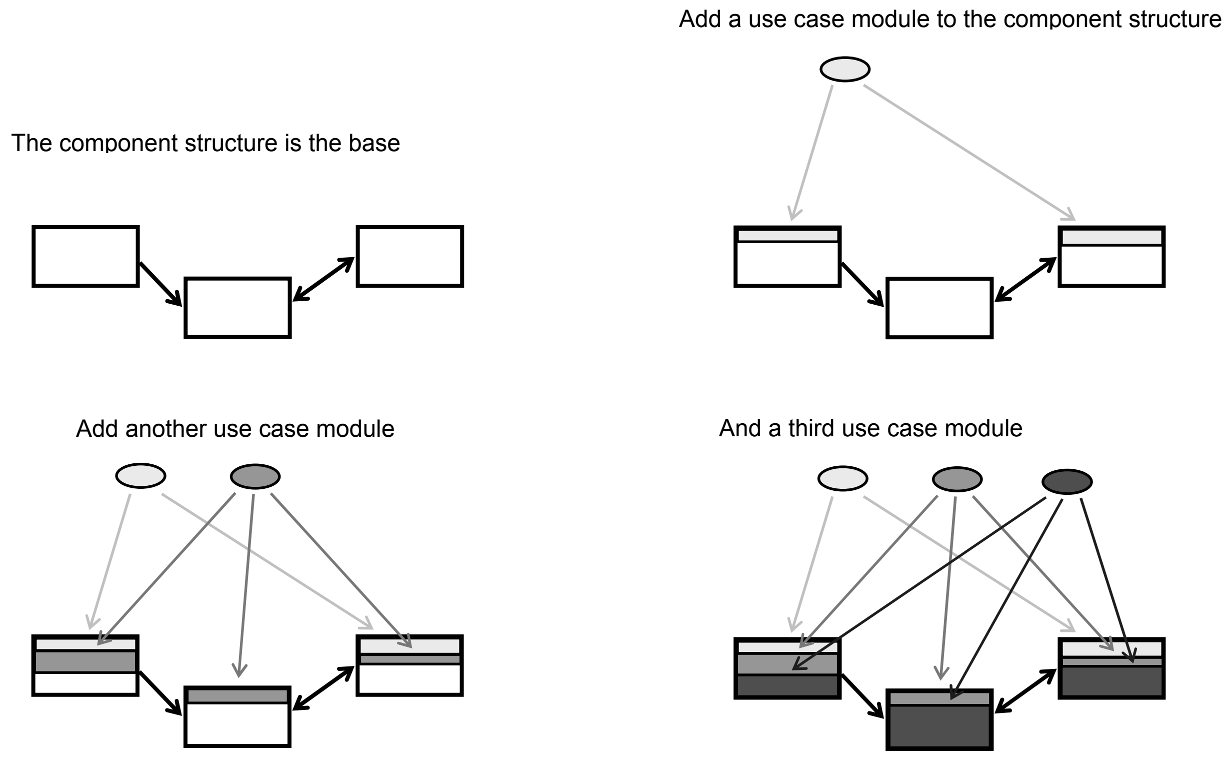
Add a use case module to the component structure
The component structure is the base
Add another use case module
And a third use case module

Figure 4: Adding Use-case Modules on Top of the Component Modules

Unfortunately, the idea was not adopted at that time because it was too similar to a patching technique (I always had to apologize for this similarity before explaining the idea). The patent application (referred to above) was not approved because there was already a patent for patching, and my proposal would have infringed on it.

In the OOSE [6] approach, we continued to support extensions in requirements and analysis, and we showed how they could be implemented in traditional object-oriented programming languages. The *Reuse* book [10] elaborated on variation points as a generalization of extension points. Many of these ideas have been carried over to the Reusable Asset Specification (RAS).[11]

The first serious attempt to implement extensions was in the development of a new generation of switches at Ericsson in the early ’90s. Extensions were taken into a new development environment called Delos which supported extensions all the way down to code.[10]

Was this a ground-breaking new idea? Obviously not, it relied on patching techniques that had been known for decades. *What was new, though, was the realization that “patching done right” is the most natural way to understand complex systems.*

Like a “Gift from Above”—Aspect-Oriented Programming

Humans generally make abstractions from an understanding of concrete things. Working the other way around, to make something that is only an abstraction into something “concrete” requires almost a religious conviction to have it be adopted. That is, you need to preach it and hopefully you make your abstraction believable. I’ve had to do this many times over the years. It was nearly impossible to make components and interfaces concrete when assembler was our implementation environment—but we did it. Modern languages like Java and C# concretized these abstractions (components and interfaces), so we didn’t have to “preach” any more. Making the extension mechanism concrete has been a challenge given that we had no programming language support for it. Even if the UML now supports it for use-case modeling, many authors bluntly recommend that it not be used. *With AOP we have the tool to explain extensions concretely, since they can be supported all the way down to code,* with languages like AspectJ [2]. Even if this is only one part of what we need, it changes a lot.

To recapitulate: **to be able to separate and later compose use cases all the way down, from requirements to executables, we need to be able to deal with peer use cases and extension use cases**. Extension use cases have been suggested for more than twenty years and are now supported with AOP. Peer use cases with overlapping behavior have so far been managed manually by developers. AOP specifically works on MDSOC [5] (Multi-Dimensional Separation Of Concerns), and HyperJ [3] allows separation of peer use cases.

Interest in AOP has grown substantially since I heard of it in Gregor Kiczales keynote speech at OOPSLA’97 (see [12]). Prior to that, in 1991, I learned of Karl Lieberherr’s work on The Law of Demeter and propagation patterns, [13] and in 1994 visited IBM in Yorktown Heights and met Harold Ossher and Bill Harrison and learned about their work on subject-oriented programming.[14]

Going back to my OOPSLA’86 paper, the base constructs have their equivalents in AOP. The mapping in Table 1 was originally made by Karl Lieberherr in a private correspondence:

Table 1. Mapping of Constructs in the OOPSLA'86 paper to AOP Equivalent

OOPSLA'86 paper	AOP Equivalent
existion	base program
extension	aspect
extensions doesn't change the base	base oblivious of aspect
extension points	join points
list of extension points	set of join points

In a moment I'll explain more.

4 PART II: TODAY—WORKING WITH USE CASES

To understand what AOP can do for us, we first need to understand state of the art of use case driven development. **Use case driven development assumes that software development is model driven**. *In its simple form it has the following sequence of models: use case to design to implementation.*

Within each iteration of the software lifecycle the team goes through the following sequence of activities:

1. find the use cases and specify each use case
2. design each use case
3. design and implement each component, and finally
4. test each use case

Hereafter I'll use the term *component* as a generic term to represent implementation elements such as classes, subsystems, and physical components.

Usually, each of these four activities represents a job taken on by a team member. Apart from the activity 3 (designing and implementing each component), all the activities are use case based; thus the term *use case driven development*.

There is, of course, more work to be done within an iteration, for instance architectural analysis and design, but for this paper we don't need to go further.

During these activities we develop the following key artifacts: use cases, use-case realizations, and components.

Use Cases

Intuitively, a use case is a sequence of actions performed by the system to yield an observable result of value to a particular user.

Formally, a *use case* is a class-like construct that describes a related set of usages of the system by a particular actor (user) type.

Use cases can be concrete or abstract. Concrete use cases can be instantiated. For example, assume the use case Make Telephone Call is an abstract use case; a concrete use case could be Make Telephone Call Using Protocol X. When a subscriber makes a call, an instance of the latter use case would be created.

The *use-case model* contains actors and use cases, and the relationships between them. It's a kind of requirements model. To make the use-case model simple, a language constraint has been enforced on the kind of acceptable relationships between use cases. *The goal is to be able to separate concerns: [1] each use case represents a concern of a set of stakeholders*. Thus, relationships between use cases as class-like things are the only acceptable relationships.

For this paper, there are only two such relationships of interest: the *generalization* relationship, which relates a concrete use case to an abstract use case; and the *extend* relationship, which adds behavior to a base use case at a set of extension points without changing the base use case. The added behavior is specified in the extending use case. In the base use case an *extension point* unambiguously references a point in the use-case description, possibly by using *before* or *after* qualifiers. At this point the *extension behaviour* specified in the extending use case will be inserted when the use case is interpreted.

Use-Case Realizations

The use-case model is an external perspective of the system—it does not represent the internal building blocks. The internals of the system are introduced in the *design model*[1]. Each use case in the use-case model is realized by one *use-case realization* in the design model. A use-case realization is a UML *collaboration* describing (e.g., using sequence diagrams) which components participate, how they interact, and what responsibilities they take on to realize the use case. *Since each use case is a different concern in the use-case model, each use-case realization is a different concern in the design model*. The realization of a use case touches many components (scattering), and a component contains pieces of several use-case realizations (tangling).

Figure 5 demonstrates scattering and tangling by modeling an ATM system with three use cases (different concerns). Each use case is designed, and the result is a use-case realization. Finally each component that participates in realizing the use cases will be designed, implemented, and unit tested.

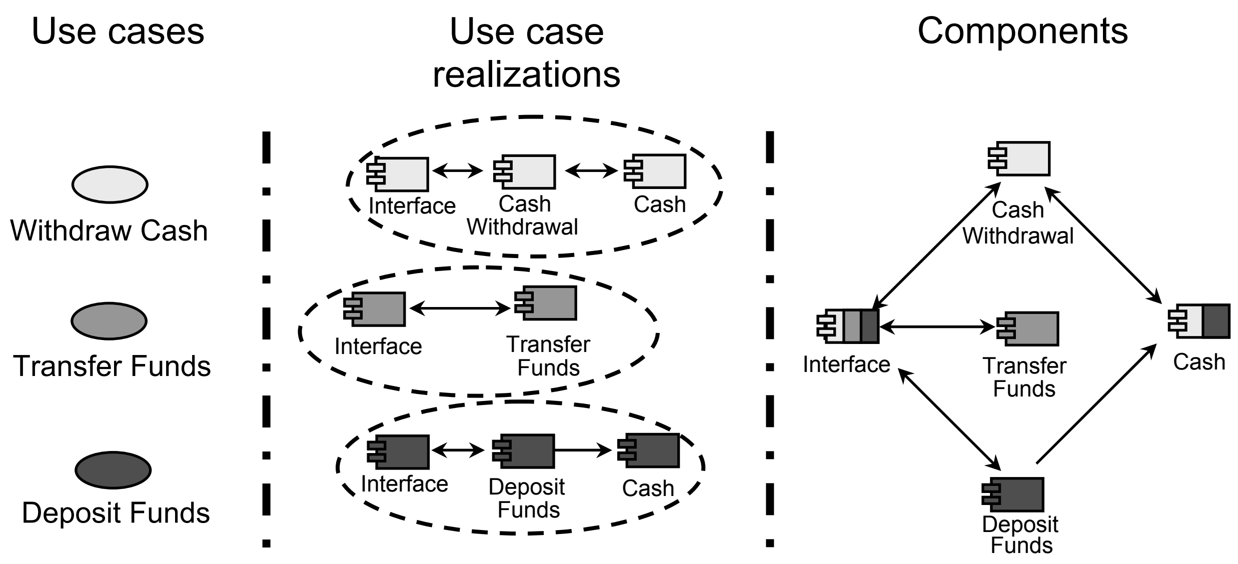

Figure 5. Scattering and Tangling

Scattering results from a use-case realization touching several components, and tangling from a component containing intermingling pieces of several use cases[10].

The generalization mechanisms between use cases are propagated to collaborations in the UML. Thus a use-case realization can reuse a more abstract use-case realization.

However, the extension mechanisms provided between use cases didn't make it to collaborations; I simply couldn't make a case for this since we had no mainstream programming language supporting the implementation of extensions as we now will have with AOP. Consequently, it is not possible to separate extension use cases from base use cases in design and implementation. The realization of the extension use case has to be dissolved into the realization of the base use case, and the base use case cannot be oblivious of the extension use case. So we do not have a fully seamless transition from use-case modeling to design—*realization of extension use cases has to be intermingled with the realizations of base use cases.*

Components

A tool can, based on the use-case realizations, generate a specification for each component by collecting all the responsibilities assigned to the component over all use cases. It's quite straightforward since the responsibilities come directly from the use cases, which the component participates in realizing.

[10] Figure 4 shows scattering and tangling, however, it doesn't show the overlap between peer use cases. It doesn't show that the Cash component has overlapping pieces from Cash Withdrawal and Deposit Funds, or that the three use cases overlap in the Interface component. If use cases didn't overlap our problem would be easy to solve. But, the overlap has to be resolved.

Now the component owner has to compose and implement all the responsibilities into a consistent whole. Issues to deal with are (1) how to reconcile the different needs of the use cases, for instance, reconciling overlapping operations, and (2) conflicts, for instance those due to concurrency (e.g., deadlock). Finally each component is unit tested against its specification.

A major problem today is that traditional languages don't support separation of concerns so the impacts of the different use cases on a component can't be kept separate—the slices can't be maintained.

5 PART III: TOMORROW—WORKING WITH USE CASES AND ASPECTS

By using aspects our approach will change substantially. We will be able to keep use cases separate all the way down to executables.

However, let's first revisit our key constructs and how they relate to AOP.

Use Cases

A *use case* corresponds most directly to a *concern*. It is a *crosscutting concern* [15] because it crosscuts the components that realize it. Concern is not a language construct in AOP, but rather pragmatics to motivate aspects.

Like components, use cases can be structured in layers according to some criteria: application specific (for instance money management in a banking system), application generic (for instance reusable banking frameworks), middleware, or systemware. We usually think of use cases as being related to applications, belonging to the application-specific or application-generic layers. However, lower-layer, or infrastructure, software can also include use cases; such use cases are concerned with distribution, persistence, debugging, performance monitoring, auditing, and so on.

There is no construct corresponding to a use case in AOP, and there shouldn't be one since AOP is a programming technique. However, AOSD and more specifically MDSOC [5] and HyperJ [4] have constructs that correspond to a *use-case module. A use-case module contains a use-case realization and a set of slices of components participating in realizing the use case; each component slice includes the design and implementation (the code) of the part of the use case that the component realizes. The corresponding construct in MDSOC is the* ***hyperslice*** *construct.*

Use-Case Realizations

The implementation of a use case is typically allocated across the system to many components, and is modeled in a use-case realization.

A *use-case realization* is a modular unit of some kind in the design model that crosscuts some components of the system's implementation. In the context of AOP it is

- either a part the *base program*; this is the case for those peer use cases that are the base for realizing the dominant decomposition,
- or an aspect; this is the case for the realization of use-case extensions.

An *aspect* is a "modular unit of crosscutting implementation." An *extension* in the OOPSLA'86 paper is directly mapped to an aspect.

The UML now needs to add the support of extensions to use-case realizations (UML collaborations) in the UML. Extension points will propagate from use cases to also be prevalent between use-case realizations (UML collaborations).

Components

Moreover, we will also need to add the support of extensions to components in the UML. This is easy to accept given that we now can get efficient support for extensions in our programming environment. Extension points will propagate from use cases via use-case realizations to all kinds of components (classes, subsystems, physical components, etc.) that participate in realizing a use case.

Some use-case realizations will need many extension points in different components or in the same component. For instance, extending a base use case with debugging traces may mean that all the components that participate have to be extended and thus we need many extension points in many components.

Extension Points

Extensions and extension points have their counterparts in AOP:

1. An *extension point* is a ***join point*** in AOP. A *join point* is a well-defined point in the execution of a program. Not every execution point would be a valid join point; some restrictions are necessary to make sure that good programming practices are applied.
2. A *set of extension points* attached to a use-case realization and its participating components is a *set of join points* in AOP. They are defined in a *join-point model* [16] which includes a mechanism (e.g., using regular expressions) to select a set of join points. This is an important contribution to the use case driven approach, something that was lacking in my 1986 paper.
3. An *extension behavior* (or a use-case fragment) is an ***advice*** in AOP. An *advice* is the code to be executed at each join point defined in the join-point model. There are three ways of executing an advice: before, after, and around the join point. An extension behavior/advice executes in the context of the extended element/join point.

A first step to get to "Tomorrow" is to add what's needed to the UML: among other things, propagate the extend relationship with extension points from use cases to collaborations, classes, operations, and so on—in fact, to every significant language feature in the UML.

What We Get with Aspects

A new kind of module (a special case of hyperslices in MDSOC) will arise: use-case modules. As discussed earlier, we need two kinds of modules: use-case modules and component modules. The use-case modules will crosscut the component modules. The two types will live side by side and play different roles. *The component modules provide the static structure of the system; the use-case modules provide the dynamic behavior added to this structure.* A developer can choose the perspective from which to view the system: either the use-case perspective or the component perspective, and work from there. The programming environment will propagate changes from one perspective to another.

Being able to keep use cases separate we would still do the activities discussed above in *Part II: Today*: (1) find the use cases and specify each use case, (2) design each use case, and (4) test each use case. But (3), design and implement each component, would be replaced by: (3a) code each use case, and (3b) compose the use case slices of each component. Since (2) and (3a) would be work carried out by the same individual—the use-case designer—the resulting sequence of activities would become:

1. find the use cases and specify each use case
2. design and code each use case
3. compose the use case slices of each component
4. test each use case

The component owner's work will change quite substantially: although she or he will not code or test the component (coding will be done by the use-case designer, and testing will be integrated with testing the use case), the component owner will still have to reconcile the different slices of peer use cases. In the future, I expect this activity to be reduced through new tooling and through collaboration between the concerned use-case designers. The programming environment will weave together the crosscutting use cases within each component into a consistent whole; this whole will depend on when the weaving occurs—the whole may be a precompiled component, an executable component, or an executing component. Furthermore, the component owner will still need to be responsible for the integrity of the "data" (the lower-level classes or attributes) of the component.

This will streamline the software development process by reducing or potentially removing the disruption of having to go via the separate activity of component design, implementation, and test**. If we can remove the component work entirely the four activities will be reduced to only two:**

1. find the use cases and specify each use case
2. design, code, and test each use case

Thus, each iteration will be worked through—use case by use case—all the way down to code (and maybe also dynamically in runtime). Right now, I don't know how far we can get, but I'm confident that we will substantially cut costs in software development and maintenance.

6 THE FUTURE

From what I have said so far, it is evident that I view AOP and AOSD as very important contributions to the software world. **Being able to support extensions all the way down to code, and later compose them, will give us a tremendous boost in all measures: cost, quality, and time.**

Regarding the future further down the road, I think that once we have committed ourselves to this new technology, there is an even bigger step just ahead of us, and we will begin to see a new era in software development, an era we could call *extensibility from the beginning,* since software will be inherently extensible.

First, as we all know, "system development is a process of progressive change" (see reference [6]). For more than a decade I have been more explicit and said "the system development process is a change process: changing from 'something' to 'something else,' and the first development step is just a special case of changing from 'nothing' to 'something.' " The future relies on this view on software. The following statements attempt to characterize this future:

- **Software is built in extensions**[11]**;** even the first build is an extension—an extension of the null existion. Thus we will downplay the term base program or my term existion, and instead we will view a base program/existion as a set of extensions[12].
- **Extensions will be of several kinds**, such as
    - new use cases or changes in use cases due to new business requirements or new features desired by users
    - platform or infrastructure changes
    - architectural, refactoring, or other improvements
- **System development will be organized as successive extensions**; this will make the system easier to understand, grow, shrink, and maintain; the cost, quality, and time measures will dramatically improve.
    - To be more precise, I mean successive extensions on top of or beneath an existion.
    - The existion is itself a set of extensions on top of or beneath a previous existion. This recursion stops when the existion is the empty set[13], something I will return to in a moment.
    - "On the top of" means extensions that add higher-layer features (application use cases). "Beneath" means extensions that add lower-layer features (infrastructure use cases).

[11] Another term would be increment.

[12] Mathematically: If Si is system release i, and Ei is extension i, then S0 = null; S1 = S0 + E1 = E1; S2 = S1 + E2 = E1 + E2; S3 = S2 + E3 = E1 + E2 + E3. Thus each Si is a summation or union of Ei. Hence, a base program/ existion is a set of extensions.

[13] Already back in 1978, after having conducted the case study, I summarized that more than 80% of all the software in a telecom system could be designed as simple extensions to a base system. With the more elaborate techniques now discussed in AOP, this number could grow much larger.

- **Extensions will be composed at some point in time**, maybe as late as runtime, to provide an integrated behavior by the system.
- **Extensions will occur during the entire system lifecycle**: over all releases, iterations, and builds.
- **Extensions must be implemented without disrupting the operations of a deployed system.**
- **The semantics of extension-based software as outlined here should be carried all the way down to the runtime environment via the code.**

In fact, I believe the programming language should define the meaning of changes and not just leave this fundamental property of software to be dealt with by the vendors of operating systems as is the case today. In my thesis [17] I defined a small demo language that was inherently extension based. Semantically, a system instance was created empty (the empty set) and then extended with use-case realizations with participating classes. The system instance went into execution mode when its use-case realizations became instantiated and thus instances of the classes were created. This is a very interesting topic but it goes beyond the purpose of this paper.

7 CONCLUDING REMARKS

Neither use case driven development nor AOP are silver bullets; they only represent two best practices. However, I believe that integrating them will dramatically improve the way software will be developed. In the short term we will be able to slice the system use case by use case over several of the most interesting lifecycle models and keep the use cases separate all the way down to the code. At some later time we will be able to recompose these slices into a consistent whole—an executing system. In the long term we will get more of extension-based software—extensions from requirements all the way down to code and runtime; and extensions in all software layers, for example, application, middleware, systemware, and extensions across all these layers.

We will get software that is easier to work with in basically all dimensions. We will get better software (higher quality) and we will—of course ☺—get it cheaper and faster.

Now it's time to get the ball rolling and get it to work.

ACKNOWLEDGEMENTS

I am very grateful for Pan-Wei Ng's in-depth review of this paper. I would also like to thank Karl Lieberherr and Harold Ossher, and my colleagues Mauro Assano, Kurt Bittner, Magnus Christerson, Gunnar Overgaard, Ian Spence, and Dean Whampler for their constructive feedback on an early version of the paper. Thanks to Catherine Southwood for editing the paper.

REFERENCES

[1] Peri Tarr, Harold Ossher, William Harrison, Stanley Sutton. "N Degrees of Separation: Multi-Dimensional Separation of Concerns". *ICSE 1999 Conference Proceedings*. pp 107-119, 1999.

[2] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. "An overview of AspectJ". *Proceedings of the European Conference on Object-Oriented Programming*, Budapest, Hungary, 18-22 June 2001.

[3] http://www.eclipse.org/aspectj/

[4] http://www.alphaworks.ibm.com/tech/hyperj

[5] H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach". *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[6] Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Overgaard. *Object-Oriented Software Engineering, A Use Case Driven Approach*. AddisonWesley, 1994.

[7] Ivar Jacobson. "Use Case Modularity". *Ericsson internal document X/Tg 2618*, 1979-12-13.

[8] Ivar Jacobson (inventor) & Ericsson (applicant), Address Sequence Variator, 1981-09-21.

[9] Ivar Jacobson. "Language Support for Changeable Large Real Time Systems". *Proceedings of OOPSLA '86*. pp 377-384, Sept 1986.

[10] Ivar Jacobson, Martin Griss and Patrik Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley, 1997. (Section 6.7.3)

[11] http://www.rational.com/rda/ras/preview/index.htm

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. "Aspect-oriented programming". In *ECOOP'97—Object-Oriented Programming*, 11th European Conference. LNCS 1241, pages 220-242, 1997.

[13] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. "Adaptive object-oriented programming using graphbased customisation". *Communications of the ACM*. pages 94-101, May 1994.

[14] William Harrison and Harold Ossher. "Subject-Oriented Programming (A Critique of Pure Objects)". *Proceedings of OOPSLA '93*, pp 411-428, 1993.

[15] Siobh´an Clarke and Robert J. Walker. “Separating Crosscutting Concerns Across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J.” *Technical Report TCDCS 200115* (Trinity College Dublin) and *UBCCSTR 200105* (University of British Columbia), 2001.

[16] M. Wand, G. Kiczales, and C. Dutchyn. “A semantics for advice and dynamic join points in aspect-oriented programming”. Work supported by the National Science Foundation under grant number CCR-9804115, January 2002.

[17] Ivar Jacobson. “Concepts for Modeling Large Real Time Systems”. Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sept. 1985.

About the author

Ivar Jacobson is a father of the following techniques: use cases, component-based development, the Unified Modeling Language, the Rational Unified Process, and business modelling with use cases and objects. He founded the Swedish company Objectory AB, which merged with Rational in 1995. He departed recently from Rational as an employee, but he is still an executive technical consultant of the company.